JuiceFinder

Fuel Your Journey with JuiceFinder –
where EV routes, charging stations, and food come together!

Chuan Li	chuanli1@seas.upenn.edu
Wei-Shen Lee	lewei@seas.upenn.edu
Zhiyang Lin	zhiylin@seas.upenn.edu
Yicheng Shen	yichengs@seas.upenn.edu

1. Introduction

As electric vehicles (EVs) continue to gain popularity, the demand for a seamless and efficient EV charging infrastructure is growing. The transition towards sustainable transportation necessitates not only a user-friendly experience for EV owners but also data-driven decision-making for governments and private sectors. This report introduces JuiceFinder, our application designed to streamline the EV charging experience and cater to the needs of both EV owners and decision-makers, ultimately fostering a sustainable transportation ecosystem. JuiceFinder simplifies the process of locating EV charging stations while simultaneously providing users with convenient access to food options nearby, making it an all-in-one solution for those on the go.

Electric vehicle owners often struggle with finding suitable charging stations, planning road trips with appropriate stops, and locating restaurants near charging stations. Our application simplifies the charging experience with features such as attribute-based charging station search, geolocation-based charging station search, road trip planning with EV-compatible charging stations, and a nearby food finder. These functionalities enable users to find the most appropriate charging stations based on specific criteria and plan their trips with ease, reducing range anxiety and ensuring access to essential amenities during their journey.

In addition to catering to the needs of EV owners, our application provides valuable insights for decision-makers through visualizations of aggregated statistics. These visualizations help analyze fuel and charging resources in relation to usage in a given region, enabling effective resource allocation and EV infrastructure growth. This data-driven approach empowers governments and private sectors to make informed decisions for the future of sustainable transportation.

In this report, we delve into the details of our application's functionalities, showcasing how it addresses the target problems and contributes to the advancement of electric vehicle infrastructure and sustainable transportation.

2. Architecture

Our application supports user authentication using Firebase authentication. We also enable login with Google or GitHub using OAuth 2.0.

We use Figma to perform multiple stages of prototyping to create a UI that is visually pleasing and user-friendly. Then, we implement the UI by utilizing and customizing MUI components. See Appendix B for our interfaces. We then translate these components to modular code in React, adding functionality to make them interactive and capable of communicating with the server.

In the API development process, we use Express.js and Node.js, to create server-side routes that handle incoming HTTP requests from the client, perform operations such as querying the database and integrating 3rd-party services, and return appropriate responses in JSON format.

We pass every free-formed address to the Google Geocoding API to obtain the (longitude, latitude) coordinate of the most relevant location. It is a fast and reliable way to locate users' input and it outperforms the Nominatim API on OpenStreetMap data which we deprecated earlier.

After we get the source coordinate and the destination coordinate, OpenRouteService API quickly provides us a basic route in GeoJSON format. The waypoints are also encoded in (longitude, latitude) coordinate form. This helps us do further route display and query design.

In the homepage, we use react-map-gl to embed an interactive map to display features such as stations and routes. It is fully isomorphic, enabling server-side rendering and ensuring best compatibility across different platforms.

In statistics page, the react-usa-map and @ant-design/plots libraries are used for the visualization of USA map, pie chart, stacked/grouped bar chart.

3. Data

In this section, we present the data sources utilized to develop JuiceFinder. We combined data from various sources to ensure accurate information on charging stations, electric vehicle specifications, food options, and vehicle registration counts.

The primary dataset used in JuiceFinder is the US Alternative Fueling Stations dataset, comprising 71,141 records and 74 columns, which helps us locate EV charging stations across the United States. Additionally, we employed the Open-EV-data dataset containing 293 electric vehicle specifications, with an average release year of 2019 (std: 1.9, min: 2011, max: 2022), ensuring compatibility between the user's vehicle and the charging stations. To enhance user experience while their EV is charging, we incorporated the Yelp dataset, focusing on food/restaurant entries in selected metropolitan areas, offering nearby dining options with an average rating of 3.55 stars (std: 0.888, min: 1, max: 5) from 150,346 rows and 19 attributes, which were later cleaned to 44,582 rows and 12 attributes.

Furthermore, we utilized two datasets related to vehicle registration counts by state: Light Duty Vehicle Registration Count and Total EV Registration Count, both covering 52 states. The Light Duty Vehicle Registration Count dataset has an arity of 12 and cardinality of 52, while the Total EV Registration Count dataset has a size of 31.7kB, arity of 2, and cardinality of 52. These datasets provided valuable insights into regional EV adoption, which helped inform decision-makers about potential infrastructure development.

For more detailed information on the datasets, please refer to the appendix section.

4. Database

Data ingestion

First, we performed exploratory data analysis on AFS dataset to discover valuable attributes and commonalities between datasets. Next, we aggregated and plotted on key features to understand the data distribution. Then, we filled and droped the null values. And we exploded some important columns which contain JSON format such as AC/DC port in OpenEV dataset. Finally, the dataset was split based on the ER diagram.

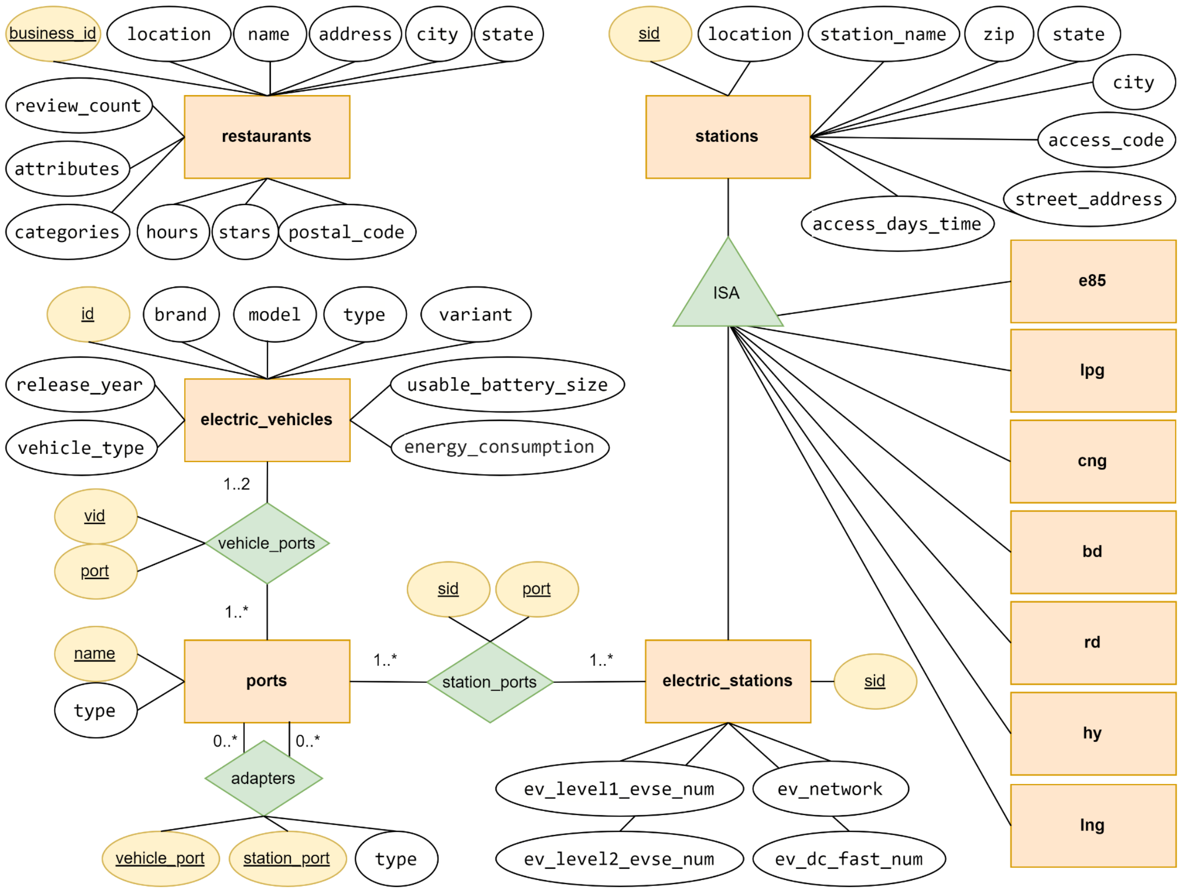
Preprocessing of Yelp dataset   Preprocessing of OpenEV dataset

Preprocessing of AFS dataset   Preprocessing of LD Vehicle registration dataset

Entity resolution

AFS dataset originally contains stations of various fuel types. We use class hierarchies to split the dataset into multiple tables using ISA. The electric stations has relation with the electric vehicle in terms of charging port compatibility. The stations port must exactly matched the vehicle port or the owner have compatible adapters on hand. The port was pulled out as a entity and two relationships formed(i.e. "vehicle_ports" and "station_ports"). Same entity set can participate more than once using different roles within a relationship set. In our case, entity set ports have two different role names called “vehicle_port” and “station_port” within adapters relationship set. Since we would like to search restaurant nearby the stations, there would be some connection on “location” attribute between two entities.

ER diagram



* Entities e85/lpg/cng/bd/rd/hy/lng are only used in statistic page to get the number of station count, hence the attributes are not listed in the diagram and will not be discussed below.

Table name	Number of instances
stations	71,141
electric_stations	59,646
electric_vehicles	277
restaurants	44,582
ports	8
station_ports	67,909
vehicle_ports	449
adapters	15

Relational Schema

1. stations(sid, location, station_name, zip, state, city, access_code, street_adress, access_days_time)
2. electric_stations(sid, ev_level1_evse_num, ev_level2_evse_num, ev_dc_fast_num, ev_network)
3. electric_vehicles(id, brand, model, type, variant, release_year, vehicle_type, usable_battery_size, energy_consumption)
4. restaurants(business_id, location, name, address, city, state, review_count, attributes, categories, hours, stars, postal_code)
5. ports(name, type)
6. stations_ports(sid, port)
7. vehicle_ports(vid, port)
8. adapters(vehicle_port, station_port, type)

Normal form and justification

1. stations: The functional dependencies are {sid -> everything else}, and sid is a superkey. For every functional dependencies {X -> A}, either A is trivial or X is a superkey. Each attribute refer to the key, the whole key, and nothing but the key, which is in BCNF.
2. electric_stations: The functional dependencies are {sid -> everything else}, and sid is a superkey. Again, it's in BCNF.
3. electric_vehicles: The functional dependencies are {id -> everything else}, and id is a superkey. Again, it's in BCNF.
4. restaurants: The functional dependencies are {business_id -> everything else}, and business_id is a superkey. Again, it's in BCNF.
5. ports: The only functional dependency is {name->type} and name is the superkey. Again, it's in BCNF.
6. stations_ports: It only contains sid and port, and (sid, port) is a superkey. Again, it's in BCNF.
7. vehicle_ports: It only contains vid and port, and (vid, port) is a superkey. Again, it's in BCNF.
8. adapters: The only functional dependency is {vehicle_port, station_port -> type}, and (vehicle_port, station_port) is a superkey. Each attribute refer to the key, the whole key, and nothing but the key, which is in BCNF

5. Complex Queries

Please see Appendix C for the complete complex queries.

The 1st query supports our application's essential functionality. It finds all stations that satisfy both location constraints and charging-related constraints. The location constraints are described in MySQL geometric format, where a polyline is constructed by a series of waypoint coordinates, and the shortest distance between a station's coordinate and that polyline is calculated by a built-in function.

The 2nd query finds desired restaurants surrounding a given location. Aside from the distance constraint, it also enables users to specify their arrival and departure time, so that these restaurants will be available upon their arrival. This is done by extracting the JSON content from the database and comparing time using built-in data type ‘TIME’.

The 3rd query is used to return info of stations within a certain distance to the input location (converted to coordinates by 3rd-party service ahead) and also satisfying a set of user defined criteria. Supported criteria include port type and charging level (at least one matches). A materialized view of the join result of three tables: stations, electric_stations, station_ports to speed up query.

The 4th query is used in the statistics page, which gets the number of stations count of various fuel types and visualizes in the bar chart. The Common Table Expression is used, and multiple joins, aggregations and unions across 9 entities are involved.

The 5th query is used by the bar chart in the statistics page to get the total number of stations across different states and port types. The Common Table Expression, aggregations, filtering, and multiple joins are involved.

6. Performance Evaluation

We utilize three critical optimization techniques to help accelerate the query of our essential filter-along-a-route feature. The combination of them make our application even more efficient on charging station searching compared to the Google Map service.

Spatial indexes are structures that organize spatial data like coordinates for efficient location-based searches. By dividing the space into smaller regions and storing references to objects within each region, unnecessary comparisons are minimized, resulting in faster search times. In summary, spatial indexes enable quicker search by narrowing down the search area and reducing comparisons. MySQL implements spatial extensions as a subset of the SQL with Geometry Types environment. This term refers to an SQL environment that has been extended with a set of geometry types. A geometry-valued SQL column is implemented as a column that has a geometry type. Each location of restaurants and charging stations is encoded as a (longitude, latitude) coordinate of geometric type POINT.

The MySQL built-in function ST_DISTANCE() accelerates the query by providing an optimized and efficient way to calculate the distance between two spatial points or a point and a polyline. It leverages spatial indexing and optimized algorithms to perform the distance calculation, resulting in faster execution times compared to manual calculations. This built-in function takes advantage of the underlying spatial index structure and eliminates the need for manual iteration and mathematical operations, leading to significant performance gains in spatial queries.

The Douglas-Peucker algorithm simplifies a polyline by reducing the number of points while preserving its general shape. It works by iteratively identifying the point that is farthest from the line segment formed by the first and last points. If this distance is below a specified threshold, the intermediate points are discarded, resulting in a simplified polyline with fewer vertices. Given a raw GeoJSON which contains 5256 waypoints in order to depict the shortest path from UPenn to MIT, this algorithm simplifies it into only 13 waypoints, while preserving the general shape that results in the same set of filtered stations as the accurate route has.

The following table illustrates how much each method optimizes the query. The input route is from UPenn to MIT. ‘Cost after optimization’ column shows the time cost when all methods are enabled. ‘Cost before optimization’ shows the time cost when one of them is disabled. For example, if spatial indexing is disabled while the other two are enabled, the query takes 106 seconds to complete, which is approximately 50 times slower than the optimum.

Method	Cost before optimization	Cost after optimization	Cost Reduction
Spatial Indexing	106s	2.3s	50x
Built-In Distance Calculator	71s	2.3s	35x
Polyline Simplification	384s	2.3s	167x

Besides, we also take the advantage of B+ tree indexing, materialized view, local caching, and client-side filtering to improve other kinds of queries. These techniques provide 1.5x-3x acceleration based on our datasets’ size and distribution. As the data becomes larger and more complex in the future, these optimization methods are expected to demonstrate more capacity.

7. Technical Challenges

Configuring Firebase authentication necessitated the incorporation of Firebase SDKs into our application and the establishment of authentication providers for Google and GitHub using OAuth 2.0. Achieving a seamless integration of these SDKs with our application architecture presented a significant challenge. Moreover, to efficiently track the current user state and provide all necessary authentication functions, we had to implement a React context. This required a thoughtful design of the context structure to ensure it effectively managed the user state, including authentication status and user data. The context also needed to expose various authentication methods, such as sign-up, login, logout, and resetting password, in a way that made them easily accessible and maintainable throughout the application. This added an additional layer of complexity to the authentication implementation process.

We performed multiple stages of prototyping using Figma and customized MUI components to finally arrive at the UI that we are satisfied with. This process presented several interconnected technical challenges for our four-person team. Ensuring a consistent design language throughout the application, such as typography, colors, and spacing, was crucial. To maintain consistency, we created multiple versions of prototypes in Figma and synchronized with each other to finalize the design elements. As a small team, effective collaboration and communication among team members were essential for the successful implementation of UI designs. We held regular meetings and discussions to review and iterate on our Figma prototypes, ensuring that everyone was on the same page regarding design decisions. Furthermore, adapting and customizing MUI components to match our Figma designs required a deep understanding of MUI's theming and customization capabilities. To tackle this challenge and ensure design consistency, we used the MUI theme and wrapped everything in a theme context. This approach allowed us to apply a uniform design language across all components in the application, achieving the desired look and feel.

Geocoding and route planning are critical to our application. The APIs which provide these services must be accurate, fast, and reliable. After careful research and thorough testing of more than 5 public geocoding and navigation API services, we choose Google Geocoding API and OpenRouteService API for their reliability and efficiency. Moreover, it requires several minutes to run queries on the raw waypoints returned by the route service. It also introduces huge burden to the client when rendering this number of waypoints and the consequent complicated polyline. Therefore, we need to do reasonable simplification to the raw route waypoints to reduce the size for better query performance. We choose the Douglas-Peucker algorithm for its generality and flexibility. We use a threshold of discarding unnecessary points based on the number of total waypoints. Combined with the built-in distance calculator, these optimization techniques reduce most route queries down to < 5s.

8. Extra Credit Features

- Log-in authentication with email/Google/GitHub
- Light/dark modes
- Extensive visualizations

Appendix

GitHub Repository: https://github.com/AdvisorChuanChuan/CIS550-JuiceFinder

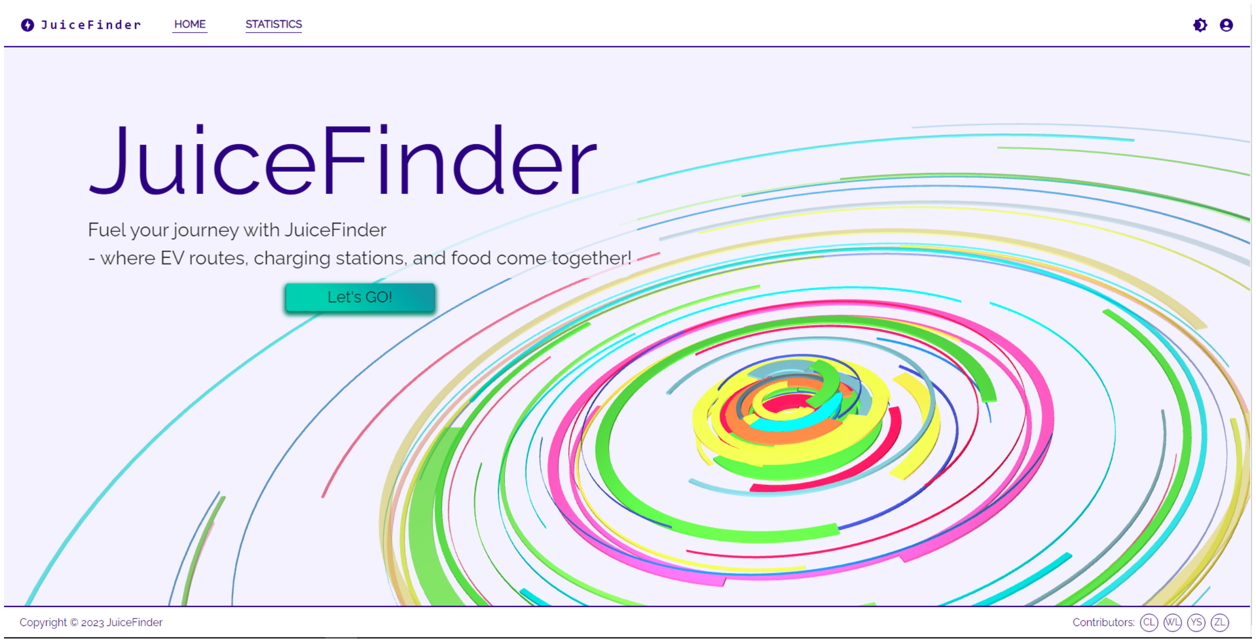
Appendix A: Dataset Details

1. **US Alternative Fueling Stations:**
    - Size: 29.9MB, 71,141 rows, 74 columns
    - Source: https://data-usdot.opendata.arcgis.com/datasets/alternative-fueling-stations/explore?location=38.572038%2C-93.984496%2C6.00
2. **Open-EV-data:**
    - Size: 318 KB, 293 JSON objects
    - Source: https://github.com/chargeprice/open-ev-data/blob/master/data/ev-data.json
    - Summary statistics:
        - Release_year: Mean: 2019, std: 1.9, min: 2011, max: 2022
        - Usable_battery_size: Mean: 40.19, std: 29.72
3. **Yelp Dataset:**
    - Size: 150,346 rows, 19 attributes, 118.9 MB (After cleaning, 44,582 rows and 12 attributes)
    - Source: https://www.yelp.com/dataset
    - Summary statistics:
        - Stars (business rating): Mean: 3.55, std: 0.888, min: 1, max: 5
4. **Light Duty Vehicle Registration Count by State:**
    - Size: 52 rows, 12 column
    - Source: https://afdc.energy.gov/vehicle-registration
5. **Total EV Registration Count by State:**
    - Size: 31.7kB, 52 rows, 2 columns
    - Source: https://afdc.energy.gov/data/10962

Appendix B: User-friendly Interfaces

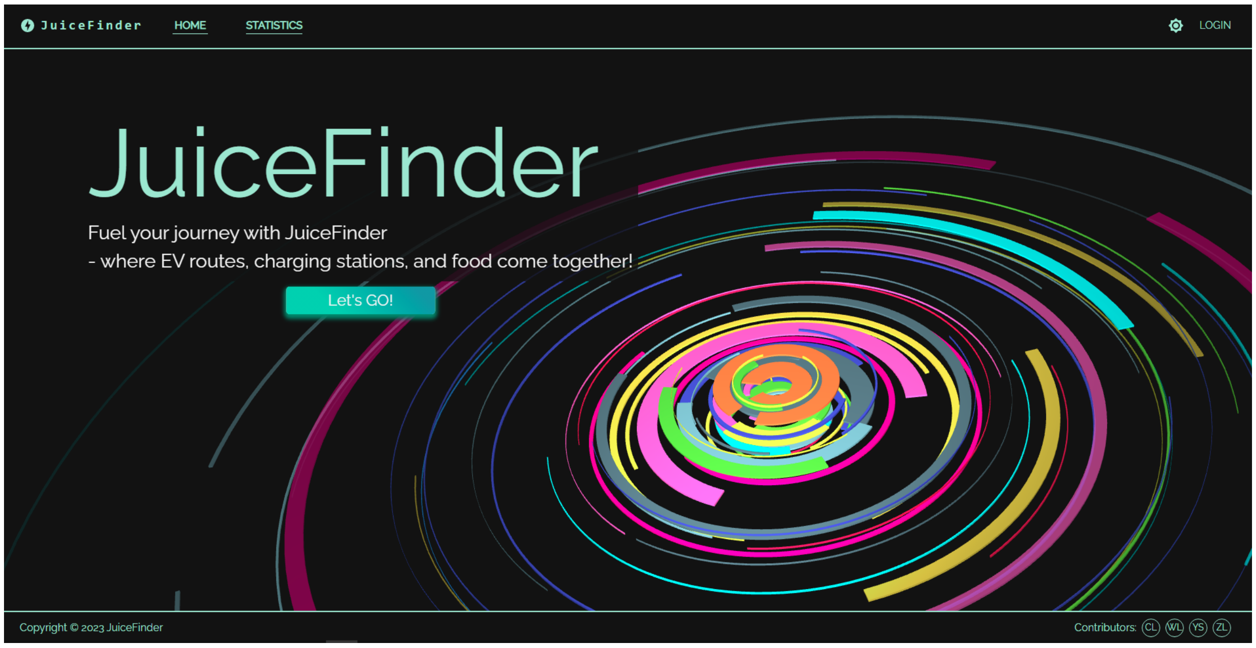
Landing page

- Dynamic wallpaper changes view based on mouse position
- ◐ ● : Light/Dark theme and login/out button
- Home: station search and path planning page (or clicking *Let's GO!* button)
- Statistics: interactive plot with valuable information provided for decision maker



Fuel your journey with JuiceFinder
- where EV routes, charging stations, and food come together!

Copyright © 2023 JuiceFinder

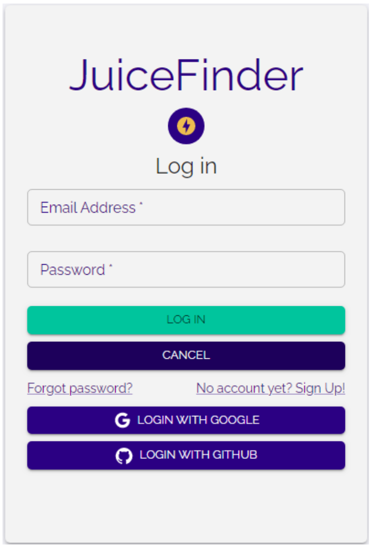


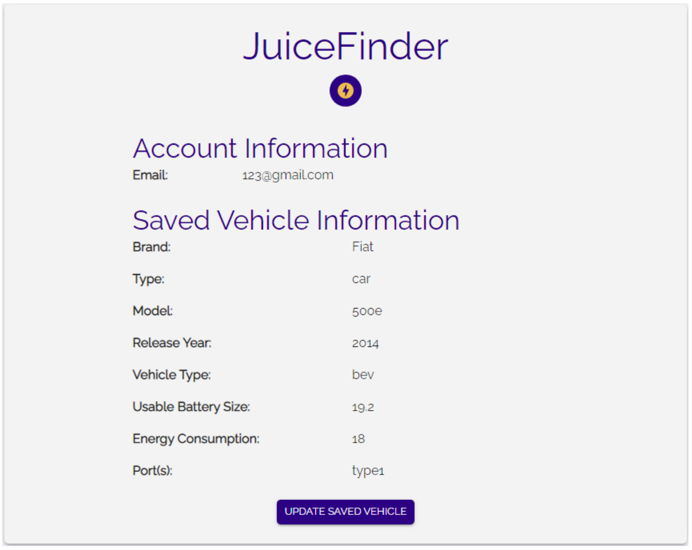
Fuel your journey with JuiceFinder
- where EV routes, charging stations, and food come together!

Copyright © 2023 JuiceFinder

User login and user settings

Support Google, Github login, or sign up new account. Easily setup vehicle information through selection menus.





Account Information

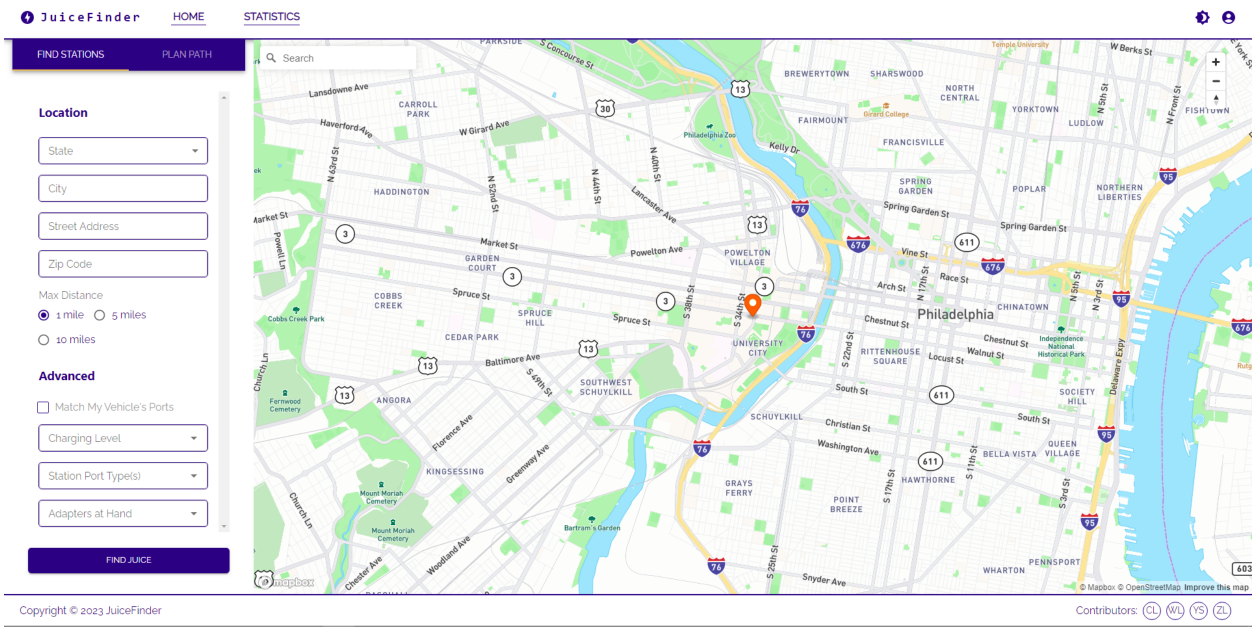
Email: 123@gmail.com

Saved Vehicle Information

Brand:	Fiat
Type:	car
Model:	500e
Release Year:	2014
Vehicle Type:	bev
Usable Battery Size:	19.2
Energy Consumption:	18
Port(s):	type1

Home/Find stations

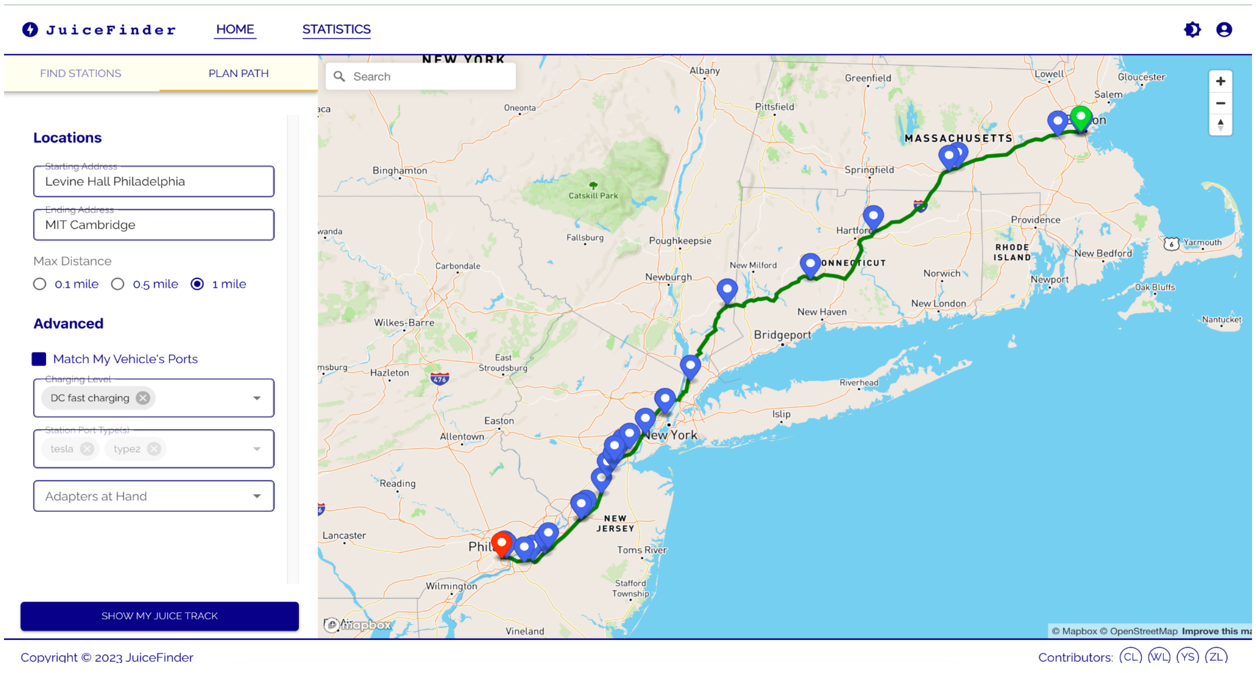
Location-based search with distance filtering. Advanced features includes vehicle's port matching, charging level and port type filtering, and adapter at hands.



Max Distance: [✓ 1 mile] 5 miles 10 miles

Home/Plan path

Planning routes based on starting and ending addresses, recommending charging stations along the routes within certain out-of-route distance. Advanced features includes vehicle's port matching, auto-filled ports based on user's vehicle settings, charging level and port type filtering, and adapter at hands.



Locations

Starting Address: Levine Hall Philadelphia

Ending Address: MIT Cambridge

Max Distance: 0.1 mile / 0.5 mile / [✓ 1 mile]

Advanced

[✓] Match My Vehicle's Ports

Charging Level: DC fast charging

Station Port Type(s): tesla, type2

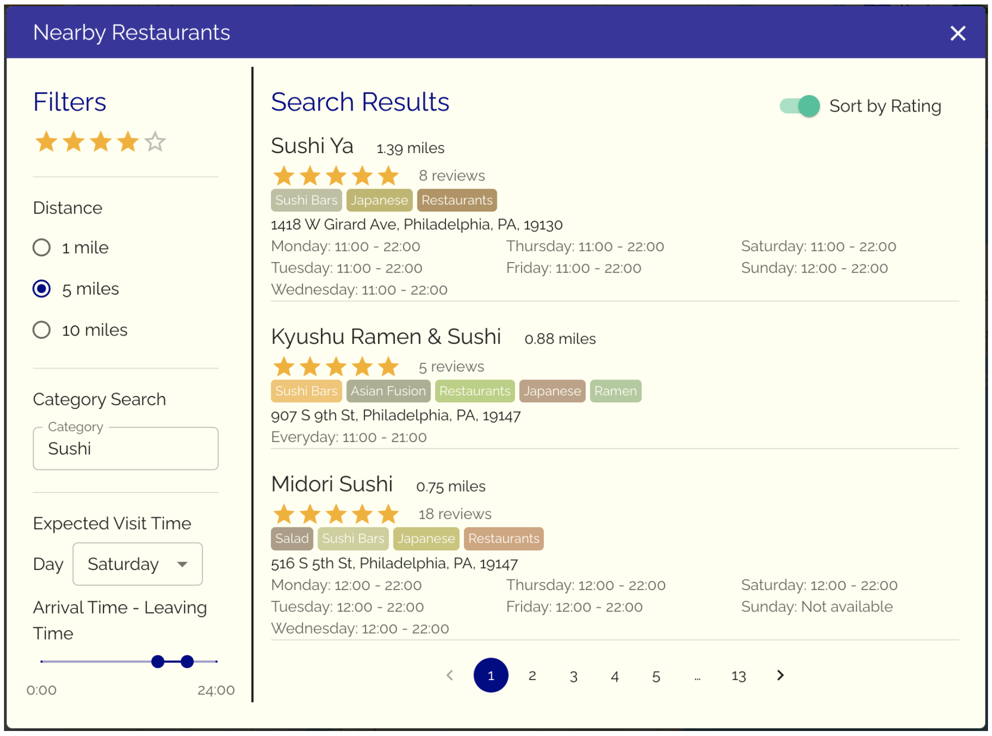
Adapters at Hand

Copyright © 2023 JuiceFinder

Contributors: (CL) (WL) (YS) (ZL)

Nearby Restaurant:

Clicking any station pins on the map and nearby restaurants will be shown. Users can filter based on distance, category, and visiting time. Sort by rating and paginations are provided.



Filters: ★★★★☆

Distance: 1 mile / [✓ 5 miles] / 10 miles

Category Search — Category: Sushi

Expected Visit Time — Day: Saturday

Arrival Time - Leaving Time (0:00 – 24:00)

Sort by Rating [✓]

Sushi Ya 1.39 miles
★★★★★ 8 reviews
Sushi Bars | Japanese | Restaurants
1418 W Girard Ave, Philadelphia, PA, 19130
Monday: 11:00 - 22:00
Tuesday: 11:00 - 22:00
Wednesday: 11:00 - 22:00
Thursday: 11:00 - 22:00
Friday: 11:00 - 22:00
Saturday: 11:00 - 22:00
Sunday: 12:00 - 22:00

Kyushu Ramen & Sushi 0.88 miles
★★★★★ 5 reviews
Sushi Bars | Asian Fusion | Restaurants | Japanese | Ramen
907 S 9th St, Philadelphia, PA, 19147
Everyday: 11:00 - 21:00

Midori Sushi 0.75 miles
★★★★★ 18 reviews
Salad | Sushi Bars | Japanese | Restaurants
516 S 5th St, Philadelphia, PA, 19147
Monday: 12:00 - 22:00
Tuesday: 12:00 - 22:00
Wednesday: 12:00 - 22:00
Thursday: 12:00 - 22:00
Friday: 12:00 - 22:00
Saturday: 12:00 - 22:00
Sunday: Not available

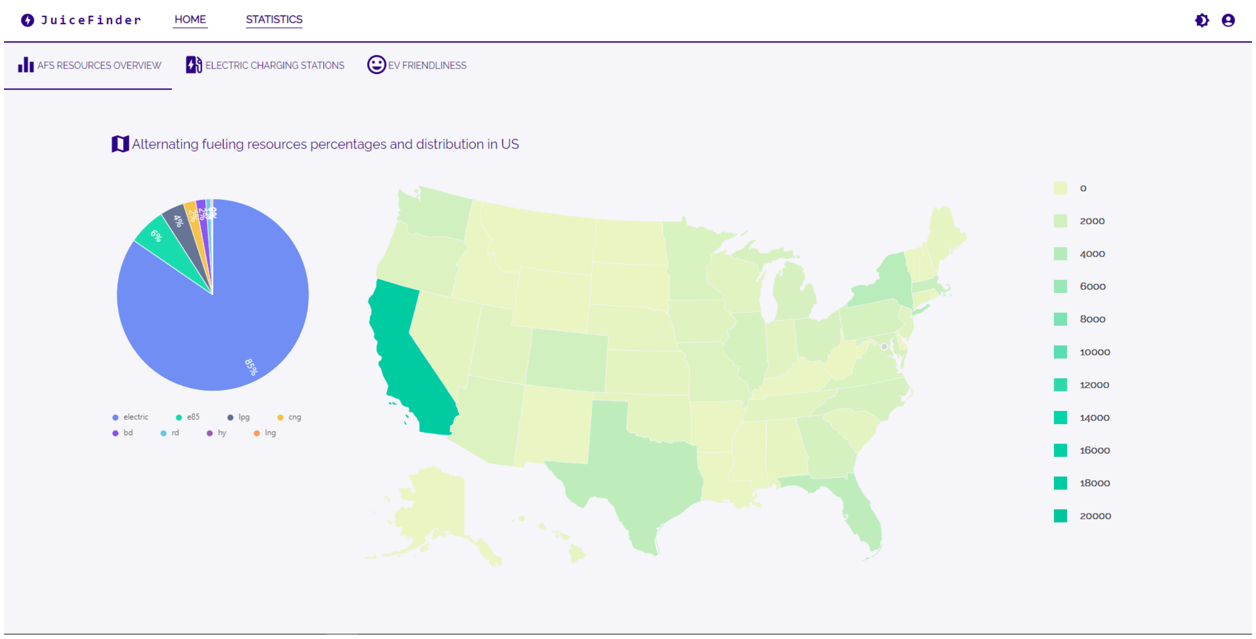
Statistic page:

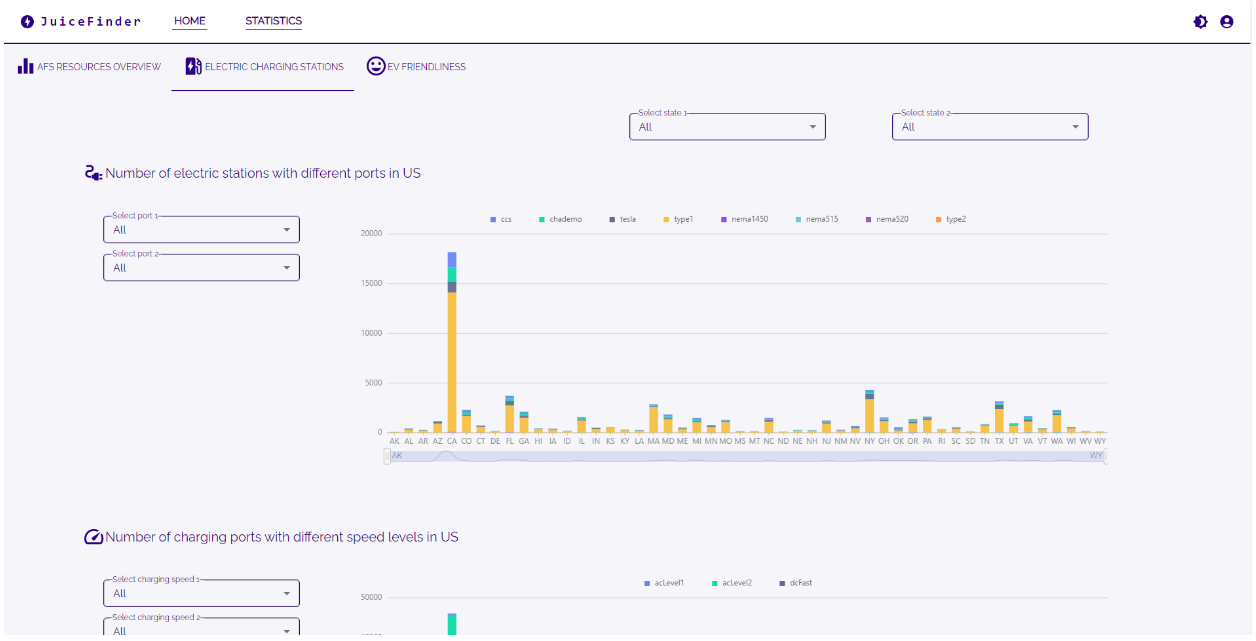
Interactive plot with valuable information provided for decision maker.

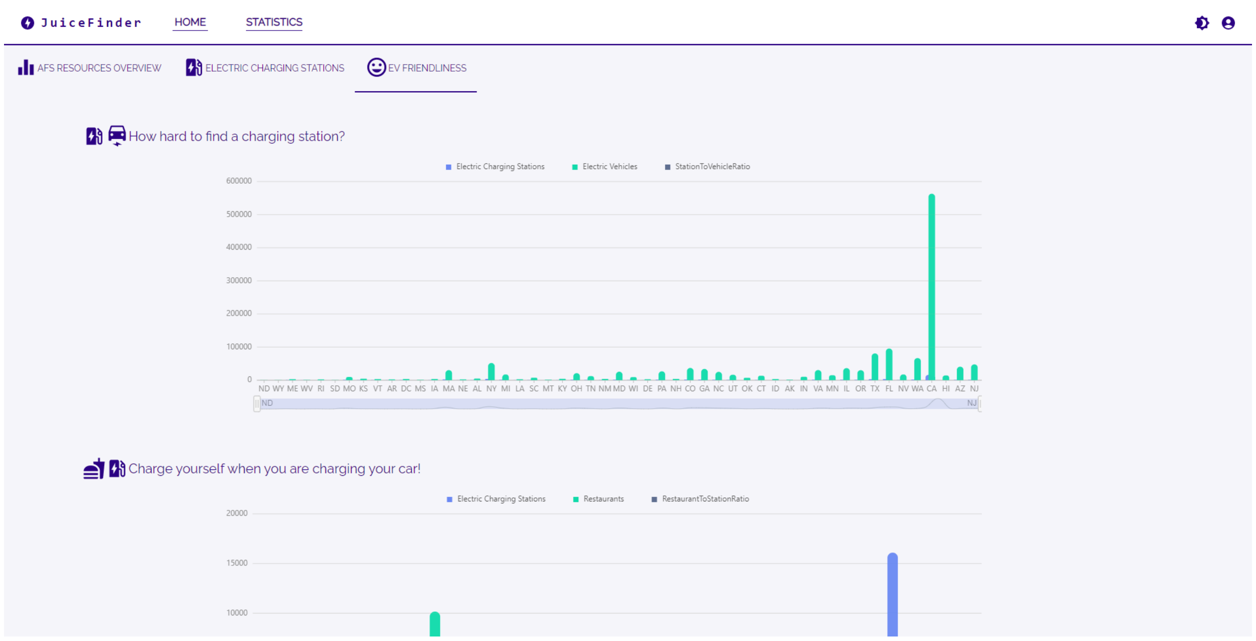
(1) AFS resources overview tab: summary of various fueling resources and vehicle count.

(2) Electric charging stations tab: number of stations group by state with different port, charging speed, and network filters.

(3) EV friendliness tab: The most EV-friendliness state ranked by two ratios.




JuiceFinder
HOME
STATISTICS
AFS RESOURCES OVERVIEW
ELECTRIC CHARGING STATIONS
EV FRIENDLINESS
Select state 1
All
Select state 2
All
Number of electric stations with different ports in US
Select port 1
All
Select port 2
All
ccs
chademo
tesla
type1
nema1450
nema515
nema520
type2
20000
15000
10000
5000
0
AK AL AR AZ CA CO CT DE FL GA HI IA ID IL IN KS KY LA MA MD ME MI MN MO MS MT NC ND NE NH NJ NM NV NY OH OK OR PA RI SC SD TN TX UT VA VT WA WI WV WY
Number of charging ports with different speed levels in US
Select charging speed 1
All
Select charging speed 2
All
acLevel1
acLevel2
dcFast
50000

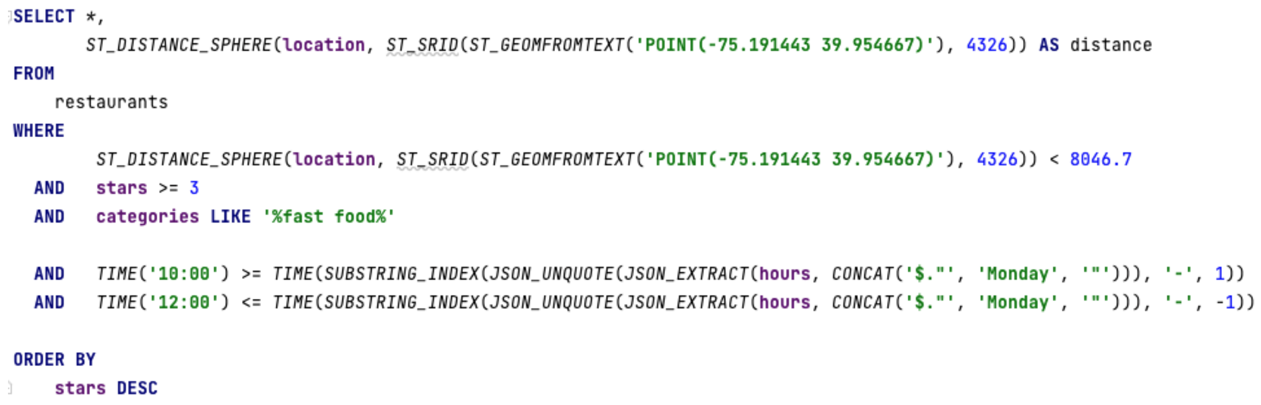

JuiceFinder
HOME
STATISTICS
AFS RESOURCES OVERVIEW
ELECTRIC CHARGING STATIONS
EV FRIENDLINESS
How hard to find a charging station?
Electric Charging Stations
Electric Vehicles
StationToVehicleRatio
600000
500000
400000
300000
200000
100000
0
ND WY ME WV RI SD MO KS VT AR DC MS IA MA NE AL NY MI LA SC MT KY OH TN NM MD WI DE PA NH CO GA NC UT OK CT ID AK IN VA MN IL OR TX FL NV WA CA HI AZ NJ
Charge yourself when you are charging your car!
Electric Charging Stations
Restaurants
RestaurantToStationRatio
20000
15000
10000

Appendix C: Complex Queries

Complex query 1:

```
SELECT DISTINCT
    sid,
    ST_Y(location) AS longitude,
    ST_X(location) AS latitude
FROM
    materialized_view_electric_stations_denorm E
WHERE
        ST_DISTANCE(
                location,
                ST_SRID(
                        ST_GEOMFROMTEXT('LINESTRING(
                            -75.15062 39.949657,
                            -75.142552 39.94646499999996,
                            ...
                            -75.15177599999998 39.901892000000004
                        '),
                        4326)
            ) < 804.67
  AND   sid IN (SELECT
                    sid
                FROM
                    station_ports
                WHERE
                    port IN ('tesla', 'type2'))
  AND   EXISTS(
                SELECT *
                FROM
                    electric_stations CLE
                WHERE
                      CLE.sid = E.sid
                  AND (CLE.ev_level2_evse_num > 0)
            );
```

Complex query 2:

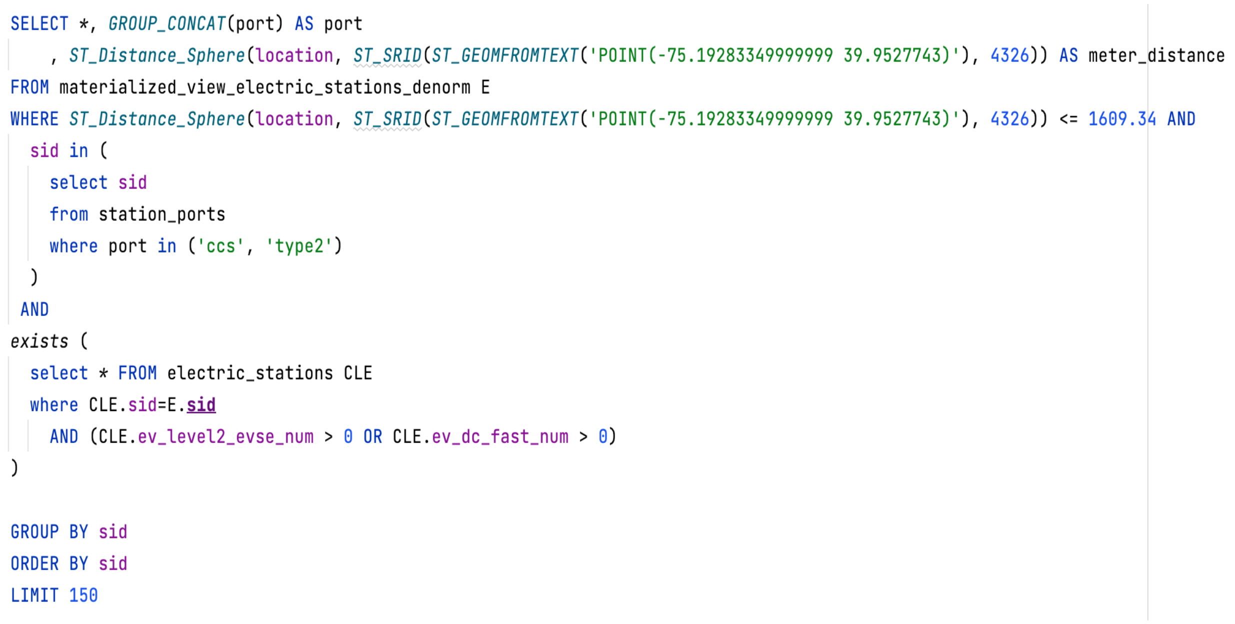


```sql
SELECT *,
       ST_DISTANCE_SPHERE(location, ST_SRID(ST_GEOMFROMTEXT('POINT(-75.191443 39.954667)'), 4326)) AS distance
FROM
    restaurants
WHERE
        ST_DISTANCE_SPHERE(location, ST_SRID(ST_GEOMFROMTEXT('POINT(-75.191443 39.954667)'), 4326)) < 8046.7
  AND   stars >= 3
  AND   categories LIKE '%fast food%'

  AND   TIME('10:00') >= TIME(SUBSTRING_INDEX(JSON_UNQUOTE(JSON_EXTRACT(hours, CONCAT('$."', 'Monday', '"'))), '-', 1))
  AND   TIME('12:00') <= TIME(SUBSTRING_INDEX(JSON_UNQUOTE(JSON_EXTRACT(hours, CONCAT('$."', 'Monday', '"'))), '-', -1))

ORDER BY
    stars DESC
```

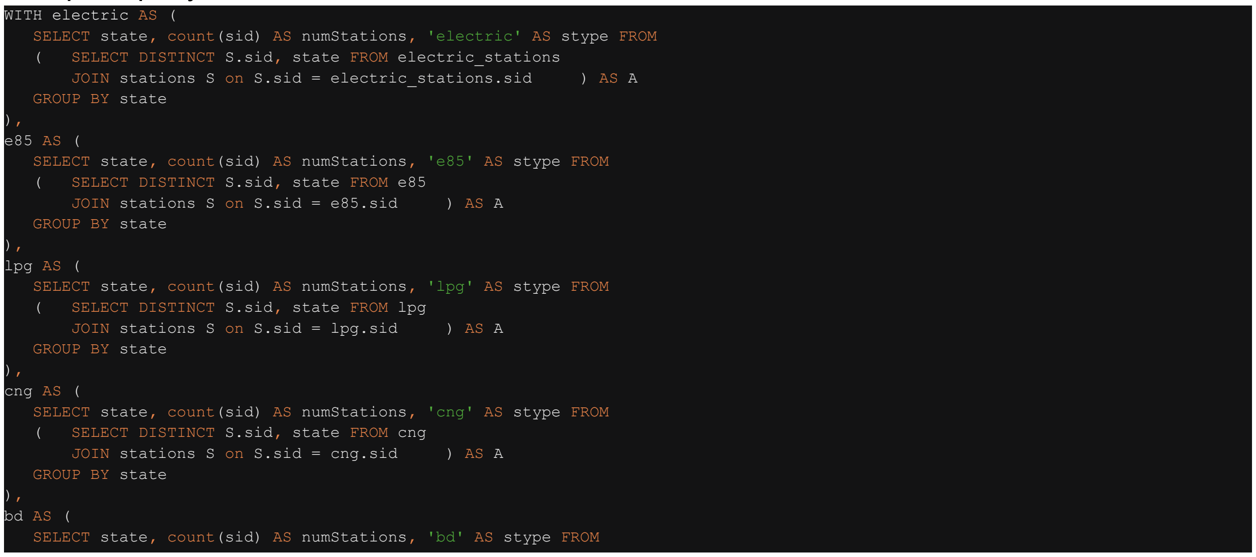
Complex query 3:



```sql
SELECT *, GROUP_CONCAT(port) AS port
    , ST_Distance_Sphere(location, ST_SRID(ST_GEOMFROMTEXT('POINT(-75.19283349999999 39.9527743)'), 4326)) AS meter_distance
FROM materialized_view_electric_stations_denorm E
WHERE ST_Distance_Sphere(location, ST_SRID(ST_GEOMFROMTEXT('POINT(-75.19283349999999 39.9527743)'), 4326)) <= 1609.34 AND
  sid in (
    select sid
    from station_ports
    where port in ('ccs', 'type2')
  )
 AND
exists (
  select * FROM electric_stations CLE
  where CLE.sid=E.sid
    AND (CLE.ev_level2_evse_num > 0 OR CLE.ev_dc_fast_num > 0)
)

GROUP BY sid
ORDER BY sid
LIMIT 150
```

Complex query 4:



```sql
WITH electric AS (
   SELECT state, count(sid) AS numStations, 'electric' AS stype FROM
   (   SELECT DISTINCT S.sid, state FROM electric_stations
       JOIN stations S on S.sid = electric_stations.sid     ) AS A
   GROUP BY state
),
e85 AS (
   SELECT state, count(sid) AS numStations, 'e85' AS stype FROM
   (   SELECT DISTINCT S.sid, state FROM e85
       JOIN stations S on S.sid = e85.sid     ) AS A
   GROUP BY state
),
lpg AS (
   SELECT state, count(sid) AS numStations, 'lpg' AS stype FROM
   (   SELECT DISTINCT S.sid, state FROM lpg
       JOIN stations S on S.sid = lpg.sid     ) AS A
   GROUP BY state
),
cng AS (
   SELECT state, count(sid) AS numStations, 'cng' AS stype FROM
   (   SELECT DISTINCT S.sid, state FROM cng
       JOIN stations S on S.sid = cng.sid     ) AS A
   GROUP BY state
),
bd AS (
   SELECT state, count(sid) AS numStations, 'bd' AS stype FROM
   (   SELECT DISTINCT S.sid, state FROM bd
       JOIN stations S on S.sid = bd.sid     ) AS A
   GROUP BY state
),
rd AS (
   SELECT state, count(sid) AS numStations, 'rd' AS stype FROM
   (   SELECT DISTINCT S.sid, state FROM rd
       JOIN stations S on S.sid = rd.sid     ) AS A
   GROUP BY state
),
hy AS (
   SELECT state, count(sid) AS numStations, 'hy' AS stype FROM
   (   SELECT DISTINCT S.sid, state FROM hy
       JOIN stations S on S.sid = hy.sid     ) AS A
   GROUP BY state
),
lng AS (
   SELECT state, count(sid) AS numStations, 'lng' AS stype FROM
   (   SELECT DISTINCT S.sid, state FROM lng
       JOIN stations S on S.sid = lng.sid     ) AS A
   GROUP BY state
),
allState AS (
   SELECT DISTINCT(state) FROM stations
   WHERE state IN ('AL', 'AK', 'AZ', 'AR', 'CA', 'CO', 'CT', 'DE', 'FL', 'GA',
           'HI', 'ID', 'IL', 'IN', 'IA', 'KS', 'KY', 'LA', 'ME', 'MD',
           'MA', 'MI', 'MN', 'MS', 'MO', 'MT', 'NE', 'NV', 'NH', 'NJ',
           'NM', 'NY', 'NC', 'ND', 'OH', 'OK', 'OR', 'PA', 'RI', 'SC',
           'SD', 'TN', 'TX', 'UT', 'VT', 'VA', 'WA', 'WV', 'WI', 'WY')
)
SELECT allState.state, IFNULL(numStations,0) AS numStations, IFNULL(stype, 'electric') AS stype
FROM electric
RIGHT JOIN allState
ON electric.state = allState.state
ORDER BY numStations desc
```

Complex query 5:

```
WITH elecStations AS (
  SELECT sid, state FROM electric_stations
  NATURAL JOIN stations
),
elecStationsPort AS (
  SELECT * FROM elecStations
  NATURAL JOIN station_ports
),
aggTable AS (
  SELECT state, COUNT(DISTINCT(sid)) AS numStations, port AS portType
  FROM elecStationsPort
  GROUP BY state, portType
  HAVING state IN ('CA','MA') AND portType IN ('type1','$type2')
),
fullTable AS (
  SELECT * FROM
  (
     SELECT 'CA' AS state
     UNION SELECT 'MA' AS state ) AS stateList,
  (
     SELECT 'type1' AS portType
     UNION SELECT 'type2' AS portType ) AS portList
)
SELECT F.state, F.portType, IFNULL(numStations, 0 ) AS numStations
FROM fullTable AS F
LEFT JOIN aggTable AS A
ON A.state=F.state AND A.portType=F.portType
ORDER BY state ASC, portType ASC
```

Appendix D: API Specificaition

Stations

Route: /stations/:id
Request Method: GET
Description: Returns all information about a station
Route Parameter(s): **id** (string)
Query Parameter(s): None
Return Type: JSON Object
Return Parameters: { **id** (string), **stationName**(string), **zip**(string), **city**(string), **state**(string), **streetAddress**(string), **location**(string), **accessCode**(string), **accessDaysTime**(string) }
Expected (Output) Behavior:

- If a valid id is provided, return the relevant information from the database as specified in the return parameters with HTTP status code 200 (OK); otherwise, return an empty object with HTTP status code 404 (Not Found). On error, return an empty object with HTTP 500 Internal Server Error

Route: /stations/electric/:id
Request Method: GET
Description: Returns all information about an electric station
Route Parameters: id (string)
Query Parameter(s): None
Return Type: JSON Object
Return Parameters: { **id** (string), **stationName**(string), **zip**(string), **city**(string), **state**(string), **streetAddress**(string), **location**(JSON), **accessCode**(string), **accessDaysTime**(string), **ev_dc_fast_num** (int), **ev_level1_evse_num** (int), **ev_level2_evse_num** (int), **ev_network** (string), **ports** ([string]) }
Expected (Output) Behavior:

- If a valid id is provided, return the relevant information from the database as specified in the return parameters with HTTP status code 200 (OK); otherwise, return an empty object with HTTP status code 404 (Not Found). On error, return an empty object with HTTP 500 Internal Server Error

Route: /stations
Request Method: GET
Description: Returns all stations in order of id, optionally paginated
Route Parameters: None
Query Parameters: **state** (string)*, **city** (string)*, **streetAddress** (string)*, **zip** (string)*, **latitude** (float)*, **longitude** (float)*, **mileDistance** (int)* (default: 10000), **accessCode** (string)*, **alwaysOpen** (boolean/key existence)*, **page** (int)*, **pageSize**(int)* (default: 50)
Return Type: JSON Array
Return Parameters: [{ **id** (string), **stationName**(string), **zip** (string), **city** (string), **state** (string), **streetAddress** (string), **location**(JSON), **meterDistance** (int), **accessCode** (string), **accessDaysTime** (string) }]
Expected (Output) Behavior:

- Return an array with all stations that match the constraints ordered by station id (ascending) with HTTP status code 200 (OK). If no station satisfies the constraints, return an empty array with HTTP status code 404 (Not Found). On error, return an empty array with HTTP 500 Internal Server Error

- If **state/city/zip** is specified, match all stations with corresponding attributes that match exactly the **state/city/zip** query parameters. If these query parameters are undefined, no filter should be applied here.

- If **streetAddress** is specified, match all stations with street addresses that contain the **streetAddress** query parameter as a substring (use LIKE to perform substring matching). If **streetAddress** is undefined, no filter should be applied for this part.

- If **latitude** and **longitude** are both defined and valid, match all stations whose distance to the point(**latitude**, **longitude**) is smaller than **mileDistance**. If either **latitude** or **longitude** is undefined, no filter should be applied here.

- If **accessCode** is specified, match all stations with access codes that match exactly the **accessCode** query parameter. If **accessCode** query param is undefined, no filter should be applied here.

- If **alwaysOpen** is specified, match all stations that are open 24/7. If **alwaysOpen** query params is undefined, no filter should be applied here.

- If **page** is defined, return all stations that match other constraints for that page number by considering the **page** and **pageSize** parameters. Otherwise, return all stations matching other constraints.

Route: **/stations/electric**
Request Method: GET
Description: Returns all stations in order of id, optionally paginated
Route Parameters: None
Query Parameters: all query params available in the route /stations/, plus: **vehiclePorts** (string)*, **stationPorts** (string)*, **adaptors** (string)*, **chargeLevel** (string)*
Return Type: JSON Array
Return Parameters: [{ **id** (string), **stationName**(string), **zip** (string), **city** (string), **state** (string), **streetAddress** (string), **location** (JSON), **meterDistance** (int), **accessCode** (string), **accessDaysTime** (string), **ev_dc_fast_num** (int), **ev_level1_evse_num** (int), **ev_level2_evse_num** (int), **ev_network** (string), **ports** ([string]) }]
Expected (Output) Behavior:

- Return an array of objects with HTTP status code 200 (OK); if no such electric stations are found, return an empty array with HTTP 404 (Not Found). On error, return an empty array with HTTP 500 Internal Server Error

- If **stationPorts** is specified, match stations that have any of the provided port types. Otherwise, no filters are applied here. Format: comma-separated with no space, e.g. "type1,ccs".

- If **chargeLevel** is specified, match stations that have ports of given charging levels. The Valid format is <level1>,<level2>", e.g. "level2,dc_fast". If **chargeLevel** is undefined, don't filter on the charging level.

Adaptors

Route: **GET /adapters**
Description: Returns all adapters available at this point.
Request Method: GET
Route Parameters: None
Query Parameters: None
Return Type: JSON Array
Return Parameters: [{ **vehiclePort** (string), **stationPort** (string), **type** (string) }]
Expected (Output) Behavior:

- Return an array of objects with HTTP status code 200 (OK). On error, return an empty array with HTTP 500 Internal Server Error

Vehicles

Route: **/vehicles/:id**
Description: Returns all information about a vehicle
Request Method: GET
Route Parameter(s): **id** (string)
Query Parameter(s): None
Return Type: JSON Array
Return Parameters: { **id** (string), **brand** (string), **vehicleType** (string) }, **model** (string), **releaseYear** (int)…}
Expected (Output) Behavior:

- If a valid id is provided, return the relevant information from the database as specified in the return parameters with HTTP status code 200 (OK); otherwise, return an empty object with HTTP status code 404 (Not Found). On error, return an empty array with HTTP 500 Internal Server Error

Route: **/vehicles**
Description: Returns all vehicles in order of brand name, optionally paginated
Request Method: GET
Route Parameter(s): None (since there are only 300+ evs currently, we will always retrieve all the evs and do the filtering on the client side. This ensures a smoother user experience.
Query Parameter(s): None
Return Type: JSON Array
Return Parameters: [{ **id** (string), **brand** (string), **vehicleType** (string) }, **model** (string), **releaseYear** (int)…}]
Expected (Output) Behavior:

- Return an array of objects with HTTP status code 200 (OK). On error, return an empty array with HTTP 500 Internal Server Error

Restaurants

Route: **/restaurants/nearby/:longitude/:latitude**
Description: Gets a list of all restaurants located within a certain distance of a given (longitude, latitude) coordinate.
Request Method: GET
Route Parameter(s): **longitude** (float), **latitude** (float) (The location's coordinate where users want to find nearby restaurants.)
Query Parameter(s):

- **maxDistMile**(float): The maximum distance threshold (in mile) on the returned restaurants to the given location.
- **stars**(float): The minimum stars of the returned restaurants
- **category**(string): The category which the returned restaurants must have.
- **day**(string): From Monday to Sunday, it is the users' expected visiting day.
- **period**(string): Specifies the users' expected visiting period. E.g. "11,14" means 11:00 to 14:00.
- **sortBy**(string): Specifies the sort key of the returned restaurants. Can be "stars", "review_count" or "distance" (default).

Route Handler: **nearbyRestaurants(req, res)**
Return Type: JSON Array
Return Parameters: [{**business_id** (string), **name** (string), **dist** (float), **stars** (float), **review_count** (int), **categories** (List(string)), **longitude** (float), **latitude** (float), **street** (string), **city** (string), **state** (string), **postal_code** (string), **hours** (JSON), **attributes** (JSON) }, …]
Expected (Output) Behavior:

- The handler gets a (longitude, latitude) coordinate and searches all restaurants within maxDistMile miles from it, filtered by the query parameters.
- If specified, the valid options for day are {'Monday', 'Tuesday', …, 'Sunday'}. Return error if the parameter is unknown.
- If specified, the standard format of period is like '9:00AM-8:30PM'. Return error if the format is wrong or the time is unrecognized.
- If specified, category should be a string of a single category.
- The valid options for sortBy are {'distance', 'stars', 'review_count'}. Return error if unknown.
- page must be specified. Return all the restaurant results on that page considering page_size.
- In the return parameters, hours should be a JSON object like {'Monday': '9:00AM-8:30PM', …}

Route: **/restaurants/:restaurantID**
Description: Gets all the information of the restaurant specified by restaurant ID.
Request Method: GET
Route Parameter(s): **restaurantID** *(string)*
Query Parameter(s): None
Route Handler: **restaurant(req, res)**
Return Type: JSON Object
Return Parameters: {**business_id** (string), **name** (string), **stars** (float), **review_count** (int), **categories** (List(string)), **longitude** (float), **latitude** (float), **street** (string), **city** (string), **state** (string), **postal_code** (string), **hours** (JSON), **attributes** (JSON) }
Expected (Output) Behavior:

- In the return parameters, hours should be a JSON object like {'Monday': '9:00AM-8:30PM', …}

Routes (Paths)

Route: **/paths/geocode/:fullAddress**
Description: Converts a full address into its corresponding longitude and latitude coordinate.
Request Method: GET
Route Parameter(s): **fullAddress** (string)
Query Parameter(s): None
Route Handler: **coordinate(req, res)**

Return Type: JSON Object
Return Parameters: {**longitude** (float), **latitude** (float)}
Expected (Output) Behavior:

- The input address is recommended to align the format as 'street name, city, state'
- The handler gets a full address string. Use OpenStreetMap Nominatim API to return a coordinate. The actual address returned by the API should be the most resemblant to the user's input. Return error if the API has no search result.

Route: **/paths/stationsNearPath/:startAddress/:destAddress**
Description: Gets a start and a destination coordinate specified by longitude and latitude. Plan a driving route and return.
Request Method: GET
Route Parameter(s): **startAddress**(string), **destAddress**(string) (free-formed address string)
Query Parameter(s):

- maxDistMile(float): The maximum distance threshold (in mile) on the returned stations to the route between start and destination.
- accessCode, alwaysOpen, …, stationPorts, chargeLevels: Same parameters in /stations/ route that describe the charging stations' attributes.maxDistMile(float): The maximum distance threshold (in mile) on the returned stations to the route between start and destination.

Route Handler: **route(req, res)**
Return Type:

- waypoints: a list of waypoints in (longitude,latitude) coordinate format.
- stations: a list of filtered stations in {sid, longitude, latitude} format.

Return Parameters: **route** (GeoJSON)
Expected (Output) Behavior:

- The handler gets a start and a destination coordinate. Use OpenRouteService API to plan a route of type GeoJSON.
- For example of a long drive, the route from Union Station, Washington DC to William H. Gray III Station, Philadelphia is 147 miles long. Its GeoJSON contains 2260 coordinates along the route, approximately 0.07 mile interval between each coordinate. We can interpolate these coordinates to select fewer of them for further query efficiency. For instance, we can increase the interval to 7 miles between each selected coordinate, that is about 23 coordinates in total. This interpolating interval should depend on the maximum station-route distance defined by the user.
- Return error if the planned route from OpenRouteService is invalid.

Route: **/routes/nearbyStations**
Description: Find all electric charging stations near to a route in a certain distance
Request Method: POST
Route Body: A list of waypoints representing the route, extracted from the GeoJSON.
Route Parameter(s): None
Query Parameter(s): **maxDistMile** (float)*, (default: 10), **accessCode** (string)*, **24-7** (boolean/key existence)*, **vehiclePorts** (string)*, **stationPorts** (string)*, **adaptors** (string)*, **chargeLevel** (string)*
Route Handler: **nearbyStations(req, res)**
Return Type: JSON Array
Return Parameters: [{**ID** (int), **longitude** (float), **latitude** (float)}, …]
Expected (Output) Behavior:

- From each line segment constructed by waypoints, we formulate a query that returns all electric charging stations having point-to-line distance less than **maxDistMile** miles. Link these queries by 'OR' under 'WHERE' section.
- For these nearby stations, apply attributes filter similar to **GET /stations/electric.**

Authentication

We use Firebase Authentication to handle signup, login, and logout. Note that firebase/auth SDK handles these functionalities on the client side. So, we don't need to create endpoints for signup, login, and logout in our backend. On our backend, we add a middleware that verifies the user ID provided by the client.

Client Side:

- Create **authContext** (This allows all components inside authContext to access the centralized currentUser state)
  - States:
    - **currentUser** state: Holds the currently logged-in user object
  - Functions:
    - **signup**(email, password): Takes email and password as arguments and creates a new user using Firebase's createUserWithEmailAndPassword() method.

        - **login**(email, password): Takes email and password as arguments and logs in the user using Firebase's signInWithEmailAndPassword() method.
        - **logout**(): Signs out the current user using Firebase's signOut() method.
- Manage the authentication state: Use Firebase's onAuthStateChanged() to add an event listener that triggers whenever the authentication state changes. When the state changes, set the currentUser state to the new user object.

Server Side:

- Create a middleware
  checkAuth(req, res, next)
  Description: Verifies the Firebase ID token sent in the request's Authorization header using Firebase-admin's verifyIdToken(). If the token is valid, the request proceeds to route handlers. If the token is invalid or missing, the request is rejected with a 401 Unauthorized status.
  Parameters:
    - req: The request object (containing the Authorization header)
    - res: The response object for sending back responses to the client
    - next: A function to call the next route handler

  Expected Behavior:
    - Case 1: If the Authorization header contains a valid Firebase ID token:
        - The decoded token is attached to the req.user property.
        - The request proceeds to the next route handler.
    - Case 2: If the Authorization header is missing or contains an invalid Firebase ID token:
        - The request is rejected with a 401 Unauthorized status.

  Usage:
    - Use **checkAuth** middleware by attaching it to any routes you want to protect

```
app.get("/protected-route", checkAuth, (req, res) => {
  res.send("This route is protected");
});
```

Users

For keeping track of users' current vehicle, we create another table in mysql database with two columns, user ID (Firebase user ID which is assigned upon signup) and vehicle ID.

Route: **/users/info**
Description: Returns an object containing a user's ID and vehicle ID if the user exists in the MySQL database.
Request Method: POST
Route Parameter(s): *None*
Query Parameter(s): *None*
Middleware: **checkAuth(req, res, next)**
Route Handler: **info(req, res)**
Return Type: JSON Object
Return Parameters: { **userId** (string), **vid** (string), **brand** (string), **vehicleType** (string) }, **model** (string), **releaseYear** (int)...}
Expected (Output) Behavior:

- The handler gets a user's ID token from req.user (passed down from checkAuth middleware).
    - Case 1: If the user ID exists in the MySQL database
        - Return an object containing user ID and vehicle ID
    - Case 2: If the user ID does not exist in the MySQL database
        - Return an empty object {}

Route: **/users/updateInfo/ :vid**
Description: Updates or inserts a new row in the MySQL database with the user's ID and vehicle ID.
Request Method: POST
Route Parameter(s): **vid** *(string)*
Query Parameter(s): *None*
Middleware: **checkAuth(req, res, next)**
Route Handler: **updateInfo(req, res)**

Return Type: JSON Object
Return Parameters: { **status** (string), **message** (string) }
Expected (Output) Behavior:

- The handler gets a user's ID token from req.user (passed down from checkAuth middleware).
- The handler retrieves the new vehicle ID from the request.
    - Case 1: If the user ID exists in the MySQL database
        - Update the vehicle ID associated with the user ID in the MySQL database
        - Return an object with a status of "updated" and a success message
    - Case 2: If the user ID does not exist in the MySQL database
        - Insert a new row with the user ID and vehicle ID in the MySQL database
        - Return an object with a status of "inserted" and a success message

Statistics

Route: /stats/overview/afsByType
Description: Returns the alternating fueling station(AFS) count aggregate by fuelType, order by numStations(descending)
Request Method: GET
Route Parameter(s): None
Query Parameter(s): None
Route Handler: /overview/afsByType(req, res)
Return Type: JSON Array
Return Parameters:{results (JSON array of {**fuelType**(string), **numStations**(int)}
Expected (Output) Behavior:
- Example:
 results = [{"electric": 8}, {"e85": 7}, {"lpg": 6}, {"cng": 5}, {"bd": 4}, {"rd": 3}, {"hy":2}, {"lng": 1}]

Route: /stats/overview/afsByState
Description: Returns the AFS count aggregate by state, order by numStations(descending)
Request Method: GET
Route Parameter(s): None
Query Parameter(s): None
Route Handler: /overview/afsByState(req, res)
Return Type: JSON Array
Return Parameters: {results (JSON array of {**state**(string), **numStations**(int)}) }
Expected (Output) Behavior:
- Example: results = [{"CA", 10000}, {"PA", 9000}, {"NY", 8000},{"WA",7000},...]

Route: /stats/overview/afsByTypeState
Description: Returns the AFS station count aggregate by type and state, order by numStations(descending)
Request Method: GET
Route Parameter(s): None
Query Parameter(s): stationType(String) (default: All)
Route Handler: /overview/afsByTypeState(req, res)
Return Type: JSON Array
Return Parameters:
return {results (JSON array of { **state**(string), **numStations**(int), **stype**(String), …}) }
, where stype=["electric", "e85", "lpg", "cng", "bd", "rd", "hy", "lng"]
Expected (Output) Behavior:
- Example: /stats/overview/afsByTypeState
 results = [{"CA", 50, "electric"}, {"CA", 30, "lpg"}, {"NY", 22, "hy"}, {"WA", 18, "rd"},...]
- Example: /stats/overview/afsByTypeState?stationType=electric
 results = [{"CA", 50, "electric"}, {"PA", 40, "electric"}, {"NY", 30, "electric"}, {"WA", 20, "electric"},...]

Route: /stats/overview/vehicleByTypeState
Description: Returns the light-duty vehicle registration count aggregate by given type and state, order by numVehicle(descending)
Request Method: GET
Route Parameter(s): None
Query Parameter(s): vehicleType(String) (default: All)
Route Handler: /overview/vehicleByTypeState(req, res)

Return Type: JSON Array
Return Parameters:
return {results (JSON array of { **state**(string), **numVehicle**(int), **vtype**(String), …}) }
, where vtype=[“ev”, “phev”, “hev”, “biodiesel”, “e85”, “cng”, “propane”, “hydrogen”,”gasoline”,”diesel”]
Expected (Output) Behavior:
- Example: /stats/overview/vehicleByTypeState
 results = [{“CA”, 50, ”ev”}, {“CA”, 40, ”phev”}, {“NY”, 30, ”e85”}, {“WA”, 20, ”diesel”},...]
- Example: /stats/overview/vehicleByTypeState?vehicleType=ev
 results = [{“CA”, 50, ”ev”}, {“PA”, 40, ”ev”}, {“NY”, 30, ”ev”}, {“WA”, 20, ”ev”},...]

Route: **/stats/electricStation/searchPort**
Description: Returns the electric station count aggregate by given port and state, order by portType(ascending) and numStations(descending)
Request Method: GET
Route Parameter(s): None
Query Parameter(s): **state1**(String) (default: All), **state2**(String) (default: All), **port1**(String) (default: All), **port2**(String) (default: All)
Route Handler: **/electricStation/searchPort**(req, res)
Return Type: JSON Array
Return Parameters:
{results (JSON array of {**state**(string), **numStations**(int), **portType**(string)})}
, where port = [“All”, “type1”, “type2”, “nema515”, “nema520”, “nema1450”, “ccs”, “chademo”, “tesla”]
Expected (Output) Behavior:
- Example: /stats/electricStation/searchPort?state1=All&state2=CA&port1=All&port2=type1
 results = [{“CA”, 50, ”ccs”}, {“NY”, 40, ”tesla”}, {“PA”, 30, ”type1”}, {“WA”, 20, ”type2”},...]
- Example: /stats/electricStation/searchPort?state1=NY&state2=CA&port1=type1&port2=type2
 results = [{“CA”, 50, ”type1”}, {“NY”, 40,, ”type2”}, {“NY”, 30, ”type1”}, {“CA”, 20, ”type2”},...]

Route: **/stats/electricStation/searchSpeed**
Description: Returns the electric charging station count aggregate by user-specified charging speed and state, order by speedLevel(ascending) and numStations(descending)
Request Method: GET
Route Parameter(s): None
Query Parameter(s): **state1**(String) (default: All), **state2**(String) (default: All), **speed1**(String) (default: All), **speed2**(String) (default: All)
Route Handler: **/electricStation/searchSpeed**(req, res)
Return Type: JSON Array
Return Parameters:
{results (JSON array of {**state**(string), **numStations**(int), **numPort**(int), **speedLevel**(string)})}
, where speed = [“All”, “acLevel1”, “acLevel2”, “dcFast”]
Expected (Output) Behavior:
-Example:
/stats/electricStation/searchSpeed?state1=All&state2=CA&speed1=All&speed2=dcFast
 results = [{“CA”, 50, 35, ”acLevel1”}, {“NY”, 30, 42, ”acLevel1”}, {“PA”, 20, 31, ”acLevel2”}, {“AZ”, 10, 49, ”dcFast”},...]
-Example:
/stats/electricStation/searchSpeed?state1=NY&state2=CA&speed1=acLevel1&speed2=acLevel2
results = [{“CA”, 50, 35, ”acLevel1”}, {“NY”, 30, 42, ”acLevel1”}, {“CA”, 20, 31, ”acLevel2”}, {“NY”, 10, 49, ”acLevel2”}]
-Example:
/stats/electricStation/searchSpeed?state1=NY&state2=NY&speed1=acLevel1&speed2=All
results = [{“NY”, 50, 35, ”acLevel1”}, {“NY”, 30, 42, ”acLevel2”}, {“NY”, 20, 51, ”dcFast”}]

Route: **/stats/electricStation/searchNetwork**
Description: Returns the electric charging station count aggregate by given network and state, order by state(ascending) and numStations (descending)
Request Method: GET
Route Parameter(s): None
Query Parameter(s): **state1**(String) (default: All), **state2**(String) (default: All), **network1**(String) (default: All), **network2**(String) (default: All)
Route Handler: **/electricStation/searchNetwork**(req, res)
Return Type: JSON Array
Return Parameters:

{results (JSON array of {**state**(string), **numStations**(int), **network**(string)})}
, where network= ['All', 'Non-Networked', 'Volta', 'EV Connect', 'POWERFLEX',
'ChargePoint Network', 'OpConnect', 'SHELL_RECHARGE', 'EVGATEWAY',
nan, 'eVgo Network', 'AMPUP', 'Webasto', 'SemaCharge Network',
'UNIVERSAL', 'EVCS', 'Blink Network', 'FCN', 'Tesla',
'Tesla Destination', 'EVRANGE', 'Electrify America', 'CHARGELAB',
'LIVINGSTON', 'FLO', 'ZEFNET', 'FPLEV', 'RIVIAN_WAYPOINTS',
'RED_E', 'SWTCH', 'CIRCLE_K', 'WAVE', 'GRAVITI_ENERGY', 'FLASH',
'RIVIAN_ADVENTURE', 'CHARGEUP']

Expected (Output) Behavior:
- Example:
/stats/electricStation/searchNetwork?state1=All&state2=CA&network1=Tesla&network2=Volta
results = [{"CA", 50, "Tesla"}, {"NY", 40, "Volta"}, {"PA", 30, "Tesla"}, {"WA", 20, "Volta"},...]
- Example:
/stats/electricStation/searchNetwork?state1=NY&state2=CA&network1='ChargePoint Network&network2=Tesla
results = [{"CA", 50, "ChargePoint Network"}, {"CA", 40, "Tesla"}, {"NY", 30, "ChargePoint Network"}, {"NY", 20, "Tesla"}]

Route: /stats/friendliness/stationToVehicle
Description: Returns the electric station count, electric vehicle count, and the ratio of the station to vehicle aggregate by state, ordered by ratio(descending)
Route Parameter(s): None
Query Parameter(s): None
Route Handler: stationToVehicle(req, res)
Return Type: JSON Array
Return Parameters:
{results (JSON array of {**state**(string), **numStations**(int), **numVehicles**(int), **stationToVehicleRatio**(float)})}
Expected (Output) Behavior:
- Example: results = [{"CA", 15, 30, 0.5}, {"CA", 20, 60, 0.33}, {"PA", 25, 100, 0.25}, {"NY", 6, 60, 0.1},...]

Route: /stats/friendliness/restaurantToStation
Description: Returns the restaurant count, the electric station count, and the ratio of the restaurant to station aggregate by state, ordered by ratio(descending)
Route Parameter(s): None
Query Parameter(s): None
Route Handler: restaurantToStation(req, res)
Return Type: JSON Array
Return Parameters:
{results (JSON array of {**state**(string), **numStations**(int), **numRestaurants**(int), **restaurantToStationRatio**(float)})}
Expected (Output) Behavior:
- Example: results = [{"CA", 30, 165, 5.5 }, {"NY", 70, 20, 3.5}, {"PA", 25, 10, 2.5}, {"NY", 25, 25, 1},...]